

Computer Science Department

TECHNICAL REPORT

JOIN PROCESSING IN A SYMMETRIC PARALLEL
ENVIRONMENT

By

Dennis Shasha
Paul Spirakis

Technical Report #158

March 1985

NEW YORK UNIVERSITY



Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012

NYU COMPSCI TR-158
Shasha, Dennis c.1
Join processing in a
symmetric ...



JOIN PROCESSING IN A SYMMETRIC PARALLEL
ENVIRONMENT

By

Dennis Shasha
Paul Spirakis

Technical Report #158

March 1985

Join Processing in a Symmetric Parallel Environment

Dennis Shasha

Paul Spirakis

Courant Institute, New York University

February 1985

1. Introduction

We present and analyze a strategy for processing joins on a highly parallel computer architecture. We model the architecture as consisting of n identical processor-memory clusters interconnected by a symmetric network onto which many processors may send at once. The strategy entails partitioning each relation horizontally based on a perfect hashing function applied to a key of the relation.

The basic join algorithm consists of projecting each relation on the joining and result columns and then sending each truncated tuple to the j th processor if the hash function applied to the join columns of the tuple yields j . Processor j then performs a local join, producing the result.

We consider three variations on the basic algorithm for the case where the join columns of at least one relation do not include a key (so there will be duplicate values): combining, tagging, and smearing.

Combining is a network operation, whereby network switches filter out some of the duplicate data destined for the same processor. As one might expect, this helps when there are many duplicates.

Tagging changes the basic algorithm by having the originating processor project on the join columns only (not the result columns) on one of the relations, then send each truncated tuple to some destination processor. The destination processor sends this tuple back if it determines that the tuple's join column values are matched by some tuple in the other relation. We show that this improves performance when there are far fewer distinct join column values than join and result column values.

Smearing changes the basic algorithm by copying the tuples of one processor, say i , to several neighboring processors for, say relation S , in order to allow tuples from the other relation, say R , that would normally hash to i to hash to any of the neighbors of i .

Our analysis depends on the properties of a particular interconnection scheme (the omega network); our approach may be more generally applicable.

2. Related Work

Our work is related to research in four areas: distributed query processing, especially [CH82, GS82]; semi-join-based processing, particularly [BC81, B79]; database machines, particularly [B79, KTM84]; and parallel algorithms [BDFW83, V83]. General query optimization strategies for complex queries [GoodShmu82, JK84, Schm79] will be relevant to later stages of our work.

Because we assume a network whose speed is comparable to local memory access time and we assume that this network can be shared, our cost assumptions differ from those made in much of the research on distributed query processing. (Our assumptions are based on the environment to be provided by the New York University ultracomputer.) Moreover, our design is based on partitioning relations across all processors. In contrast, most distributed query processing research is concerned with minimizing the volume of communicated data in the absence of partitioning [BGWRR81, ES80, HY78, HY79, WY76]. Chu and Hurley [CH82] search for conditions that optimize both communication and processing costs. They

prove several theorems with strong intuitive appeal. For example, they show that in a multi-processing setting consisting of coequal processors and memories, performing unary operations (such as selection and projection) before joins reduces total costs. Unfortunately, their work implicitly assumes no partitioning so we have not been able to use their stronger results. Gavish and Segev [GS82] study the problem of minimizing communication costs for intra-relation queries such as intersection and set difference for horizontally partitioned relations. Because the queries are intra-relational, the implementation of the queries consists of combining different partitions at one site. This makes each partition play the role of a relation, again limiting the relevance of their work to ours.

Much of the early research on database machines was most successful in optimizing selection and projection by employing special purpose input/output controllers that could perform these functions as data passed through them [S79, SL75, O82]. Recently, various tree structured interconnection topologies have been proposed [GS81, SZ84] to minimize the time to communicate between processors. Research has begun on a dataflow approach as well [BD80]. In addition, several architectures have used specialized processors to perform the various functions such as projection, selection, sort, and join [BO79, D81, MH81]. These architectures put a premium on the locality of data for joins, because their interconnection networks make arbitrary permutations of data expensive. But this limits the extent to which more processors actually help, since it is always possible for a join to require a lot of communication no matter what filtering heuristic one uses. Our strategy is to use a network which offers fast communication and then to use hash partitioning to distribute the processing cost as much as possible. For many cases, this gives nearly optimal speedup (i.e. speedup proportional to the number of processors with a log factor of degradation). In our use of hashing, our work is similar to the work of Kitaregawa, Tanaka, and Moto-oka [KTM83, KTM84]. The primary environmental difference is that we use a sharable network instead of a pipelined ring network. We also analyze improvements to our basic algorithm for cases when it may not work so well.

3. Basic Algorithm

Our strategy is based on partitioning each relation by hashing on a key (called the *partition key*) of that relation. In the next section, we show that this distributes the tuples of each relation almost evenly.

3.1. Notation

In the general case, we have a join of two relations R and S on join columns C (by renaming we can assume a natural join) and projected on attributes A from one or both of these relations. The tuples of R at processor i are labelled Ri and similarly for S . Projection of one or more tuples Ti on attributes B is denoted $Ti[B]$. Projection removes duplicates. Each hash function h is a function from the values under consideration to the numbers between 0 and $n-1$, where n is the number of processors. $Rtag$ is a tag indicating that the accompanying tuple comes from the relation R and similarly for S tag.

3.2. Basic Algorithm

Basic Algorithm

- (1) At each processor i ,
 $R_i' := R_i[C \cup A]$
 $S_i' := S_i[C \cup A]$
- (2) for each $t \in R_i'$, send (t, R_{tag}) to $h(t[C])$.
for each $t \in S_i'$, send (t, S_{tag}) to $h(t[C])$.
- (3) At each processor j ,
join the incoming tuples from R and S and
produce a join result.

This basic algorithm incorporates several possibilities. For example, if $R[C]$ is the partition key of R , then none of R 's tuples need to be sent over the network. Also, if none of the attributes of R are in the result of the join, then this algorithm only sends the projection $R_i[C]$ across the network, performing a semi-join [B79, BC81] in effect.

3.3. Processing Costs

We have written the algorithm as if the join only starts after the communication is complete. Actually, we can do some processing during communication. Suppose R_{dest} is the set of tuples from R reaching some destination processor and let S_{dest} be the tuples from S reaching that processor. We can join the two relation subsets in time $O(|R_{dest}| + |S_{dest}|)$ by preprocessing the relations during the communication step. Here is how.

When a tuple from R_{dest} (respectively, S_{dest}) comes to the processor, insert it into a B+-tree [Schk82] tagged R (respectively, S) based on its C values. After all tuples have entered, intersect the C -values of the two B+-trees. This takes time $O(|R_{dest}[C]| + |S_{dest}[C]|)$, because the C -values are in sorted order at the leaves of the B+-trees. To format the output, take the cross product of the R_{dest} and S_{dest} tuples associated with each C value in the intersection.

Without pipelining in this way, the processing cost would be $O((|R_{dest}| + |S_{dest}|) \log(|R_{dest}| + |S_{dest}|))$. In a single processor, the time would be $O((|R| + |S|) \log(|R| + |S|))$.¹

4. Basic Probability Results

The performance of this algorithm and its variants depends on how hashing maps m distinct values to $0, \dots, n-1$. Since we are always concerned with bounding the maximum of this distribution from above, we use the following two results.

Equidistribution Lemma: Suppose f is a function from a set M with cardinality m to $0, \dots, n-1$, such that the probability that $f(x)$ is $i, 0 \leq i \leq n-1$ is $\frac{1}{n}$.

- (1) If $m \geq a n \log_e n$, and $a > 1$ is a constant. Then

$$\max \text{ over } i (\{ | \{ x \mid x \in M \text{ and } f(x)=i \} | \}) \leq \frac{2m}{n} \text{ with probability } 1 - 2n^{-(a-1)^3}.$$

- (2) If $n \leq m \leq n \log_e n$, then

$$\max \text{ over } i (\{ | \{ x \mid x \in M \text{ and } f(x)=i \} | \}) \leq \log m \text{ with probability bounded below by } 1 - n^{1 - (\log \log n)^2}.$$

- (3) If $m \leq n$, then

$$\max \text{ over } i (\{ | \{ x \mid x \in M \text{ and } f(x)=i \} | \}) \leq \log m + 1 \text{ with probability bounded below by } 1 - (m/n)^{\log m}.$$

¹ If C is not a superkey of either relation, the complexity could increase to $O(|R_{dest}| * |S_{dest}|)$. In that case, the complexity of joining the two relations in one processor is $O(|R| * |S|)$.

This lemma tells us that with high probability, if M is large, it will be equally distributed within a factor of 2; and if M is small, no more than $\log m + 1$ values from M will map to a single value.

Pigeoning Lemma: Suppose f is a function from a set M with cardinality m to $0, \dots, n-1$, such that the probability that $f(x)$ is $i, 0 \leq i \leq n-1$ is $\frac{1}{n}$. If $m \geq bn \log n$ for $b > 2$, then, with probability at least $1 - n^{-b+2}$, for every i , there is an $x \in M$ such that $f(x) = i$. \square

5. Partitioning based on a Key

We assume that each relation is partitioned horizontally among the processor-memory clusters by means of a perfect (see [CW77, G81]) hashing function, from a key to $\{0, 1, 2, \dots, n-1\}$. This has the effect that the probability that a tuple $t \in R$ will be assigned to any particular processor is $1/n$.

For large relations, the equidistribution lemma promises us an equal distribution within a factor of 2.

Example 1: for $n = 1000$ and $|R| = 100000$, the probability that every processor has fewer than 200 tuples is greater than $1 - 10^{-9}$.

6. Network Assumptions

We assume a packet-switched network in which a message can travel from one processor to any other in $\log_2 n$ time in an unloaded network, interconnecting n processors. Omega-style (also known as banyan-style) networks [GL73, KS83] realize this delay. See figure 1. Moreover, all processors may send a packet at one time.

Whereas our techniques are applicable to any network of this type, our analysis depends on certain specific properties of banyan networks. In a banyan network, there is one path through the network switches from any processor to any other (figure 1). Consider a path from processor i to processor j . The switch nearest j is numbered 1, the switch feeding this switch is numbered 2, and so on up to the switch nearest i which is numbered $\log_2 n$ (n is a power of 2). According to this numbering, a partial path from i up to a switch at level r may feed 2^r processors. This will be important in our analysis of the combining technique.

7. Communication over a Banyan Network

The communication cost is the sum of the cost of sending one relation at a time. This reduces the number of cases we consider, but is pessimistic (by no more than a factor of 2) in that we can in fact send both relations concurrently, if both must be sent.

We use a sending protocol in which a processor that has to send a set of tuples will send with probability $\frac{1}{2}$ in any given cycle. There are two cases. In the first case, the join columns constitute a key of the relation to be sent. This results in a nearly uniform load on the intermediate switches, giving us the following lemma.

Network Lemma 1 (equiprobable case): If each processor i sends out_i tuples and each processor j receives in_j tuples in a communication step using the above protocol, and each tuple is equally likely to go to any destination processor, then the time the communication step takes is $O(\log n + outmax + inmax)$, where $outmax$ is the maximum number of tuples leaving a processor and $inmax$ is the maximum number of tuples entering a processor. For relations whose sizes exceed $n \log n$, this becomes $O(outmax + inmax)$. \square

In the general case, C does not constitute a key, so the performance might degrade at each of the $\log n$ switch levels.

Network Lemma 2 (general case): If each processor i sends out_i tuples and each processor j receives in_j tuples in a communication step using the above protocol, then the time the communication step takes is $O(\log n * (out_{max} + in_{max}))$, where out_{max} is the maximum number of tuples leaving a processor and in_{max} is the maximum number of tuples entering a processor. \square

Example 2: Suppose we join two large relations, R and S , on C and C is an alternate key (or superkey) of both relations. Then $out_{max} = in_{max} = (2|R|/n) + (2|S|/n)$ so the communication time is $O((|R| + |S|)/n)$. The additional processing time is $O((|R| + |S|)/n)$ assuming we can pipeline the construction of the B+ trees (see subsection above on basic algorithm). This is $O(n)$ speedup over computing on a single machine.

Example 3: Suppose we join two large relations, R and S , on C and C is an alternate key (or superkey) of S , but not of R . Suppose $|R[C]| = 1$. Then the communication and processing times are both $O(\log n * (|S|/n) + |R|)$. This gives no speedup if $|R|$ is large. It is for cases like these that we consider optimizations to the basic algorithm.

8. Algorithm Optimizations

We consider three optimizations on the basic algorithm: combining, tagging, and smearing. These optimizations may be applied independently to the two relations. Thus, combining may be applied to relation R , whereas tagging is applied to relation S . However, only tagging and smearing may apply together to a single relation.

To see when to apply these optimizations, we must characterize the distinct join cases. To do so, we need four propositional variables $CiskeyofR$, $AinattofR$, $CiskeyofS$, and $AinattofS$. $CiskeyofR$ holds when the join columns C are the key that the hash partitioning of R was based on. $AinattofR$ holds when some of the target columns besides those in C are in R . That is $AinattofR$ if $A - C$ are attributes of R . $CiskeyofS$ and $AinattofS$ have analogous meanings with respect to S . Here we describe the four cases that determine how R is processed. The decisions for S are analogous.

$CiskeyofR$	use basic algorithm
$\neg CiskeyofR$ and $\neg AinattofR$	semi-join projected on S , use basic algorithm, combining may help
$\neg CiskeyofR$ and $AinattofR$ and $AinattofS$	not semi-join case, use basic algorithm if $C \cup A$ is not a superkey of R then combining may help tagging does not help in general
$\neg CiskeyofR$ and $AinattofR$ and $\neg AinattofS$	semi-join projected on R , if use basic algorithm and $C \cup A$ is not a superkey of R , then combining may help if $ R[C] < R[C \cup A] $ then tag if $ R[C] \leq \log n$ and $ S[C \cup A] \leq n^2 / (2 R[C])$ then smear and tag

Combining entails changing the network in order to reduce the number of duplicate tuples from R reaching each destination processor.

Tagging changes the basic algorithm by projecting on the join columns only instead of on the join and result columns. Tagging also adds a step to return the tuples whose C values are included in the join (see step (4) below) and one more step (5) to produce the result.

Tagging is not generally useful when $A \text{ in } S$ holds, because then tagging requires that each processor i send $R_i[C]$ tuples in step (2) and then send $R_i[C \cup A]$ tuples in step (5) of the modified algorithm for those C values that participate in the join. We don't expect to be able to predict how many of the original $R_i[C \cup A]$ tuples the join eliminates, so we cannot consider this variant to be useful.

Basic Algorithm modified for tagging R

(For illustrative purposes, we use the standard operations from the basic algorithm for S . What happens to the S tuples doesn't change what happens to the R tuples.)

- (1) At each processor i ,
 $R_i' := R_i[C]$
 $S_i' := S_i[C \cup A]$
- (2) for each $t \in R_i'$, send (t, R_{tag}, i) to $h(t[C])$.
for each $t \in S_i'$, send (t, S_{tag}) to $h(t[C])$.
- (3) At each processor j ,
join the incoming tuples from R and S .
- (4) for each (t, R_{tag}, i)
if t is in the join result then send (t, R_{tag}, i) back to i .
- (5) if $A \text{ in } S$
then {not generally useful case}
for each returned (t, R_{tag}, i)
send all tuples $t' \in R_i[C \cup A]$
such that $t'[C] = t[C]$
to $h(t[C])$
else {semi-join case}
for each returned (t, R_{tag}, i)
put all tuples $t' \in R_i[C \cup A]$
such that $t'[C] = t[C]$
in join result.

When $|R[C]|$ is small (say $|R[C]| \ll n$) and C is not the key of R , a single value from $R[C]$, say x , will tend to be distributed over all n processors. Tagging, therefore, allows the possibility that some processor j in step (3) may receive n tagged tuples from R with value x . To avoid this smearing modifies the algorithm by copying each S tuple to several processors. This allows each tagged $R[C]$ value to go to any of these several processors, reducing the build-up at those processors.

Basic Algorithm modified for tagging and smearing R
(As above, we use the standard operations for S.)

- (1) At each processor i ,
 $R_i' := R_i[C]$
 $S_i' := S_i[C \cup A]$
- (2) for each $t \in R_i'$, send (t, R_{tag}, i) to
 some processor in the range $[(h(t[C]) - k) \bmod n.. (h(t[C]) + k) \bmod n]$
 (send to each processor in turn, deterministically)
 for each $t \in S_i'$, send (t, S_{tag}) to
 all processors in $[(h(t[C]) - k) \bmod n.. (h(t[C]) + k) \bmod n]$.
- (3) At each processor j ,
 join the incoming tuples from R and S.
- (4) for each (t, R_{tag}, i)
 if t is in the join result then send (t, R_{tag}, i) back to i .
- (5) assume not(AattinS) {semi-join case}
 for each returned (t, R_{tag}, i)
 put all tuples $t' \in R_i[C \cup A]$
 such that $t'[C] = t[C]$
 in join result.

9. Analysis of the Optimizations

The objective of analysis is to produce an algorithm for deciding when to use each of the possible optimizations. The two parameters we want to minimize are *inmax* and *outmax*, since these are the values that both the communication costs and processing costs depend on. We start by introducing notation and assumptions that are common to all our analyses. Then we analyze combining, tagging, and smearing in turn.

9.1. Notation and Assumptions for Analysis

Let $R[C] = \{rc_1, \dots, rc_m\}$, where $|R[C]| = m$. Let $R[C \cup A] = \{rca_{11}, \dots, rca_{m1}\}$, where $rca_{ij}[C] = rc_i$. Finally, let v_{ij} be the number of processors containing rca_{ij} .

If all this information were available and if we knew the exact distribution of the rca values, we could arrive at exact values of *inmax* and *outmax*. However, it is infeasible to obtain this information in a real system. Therefore, we make the following simplifying assumptions called *uniformity assumptions*:

- 1) The number of distinct $R[C \cup A]$ values whose projection on C is rc_i is the same for all i and is $r = |R[C \cup A]|/|R[C]|$.
- 2) Each value in $R[C \cup A]$ is in v processors.

Crude as these assumptions are, they help us decide when to apply the optimizations.² One assumption that would be too crude would be to assume that the $R[C]$ tuples are distributed evenly over the destination processors. The reason is that $|R[C]| = m$ could be small. We define the parameter c to be the maximum number of distinct $R[C]$ values that hash to a single processor. By the equidistribution lemma, with high probability, $c \leq \frac{2m}{n}$ if $m > n \log n$, otherwise $c \leq \log m$.

² They understate the desirability of using the optimizations. For example, (1) causes the analysis to make tagging seem less useful than it could be. (2) makes combining seem less useful than it could be.

Basic Algorithm Lemma: Under the uniformity assumptions and assuming $mr \gg n \log n$, $\frac{mr}{n} \leq outmax \leq \frac{2mr}{n}$ and $inmax = cr$. \square

9.2. Analysis of Combining

A combining network tries to prevent more than one copy of the same tuple from going to any processor. In the ideal case, as soon as a tuple passes through the network, the network remembers it and eliminates every other instance of that tuple.

For concreteness, let us say we are going to apply combining to $R[C \cup A]$ whose cardinality is mr . This will only help if $C \cup A$ is not a superkey of R . In that case, tuples with the same values on those attributes should be distributed across the processors, because the partitioning is based on a key of R . According to our uniformity assumption, each $R[C \cup A]$ value is in v processors. Thus, the total number of tuples that will be sent is mr . These tuples are approximately equi-distributed across the processors. Hence we have the following lemma.

Ideal Combining Lemma: Under the uniformity assumptions above, ideal combining reduces $inmax$ from cr to cr . \square

To see how useful combining is, we should note first that since combining occurs in the network, combining will not reduce $outmax$. Combining helps significantly if $cr > outmax$. Since we can approximate $outmax$ by $\frac{mr}{n}$, provided mr is large, combining helps significantly whenever $c > \frac{m}{n}$.

Unfortunately, the network has no global oracle to eliminate duplicate values, so we now analyze a "non-ideal" implementation of combining, which approximates existing implementations. Our model is the following:

- (1) Each switch stores the q distinct values that passed through the switch most recently. If a value enters the switch that is one of those stored, the new value is removed. (It is a duplicate.) The value q is a design parameter of the switch.
- (2) Given that a value passed through a switch at least once in the past, the probability that it is stored is equal to $\frac{q}{f_i}$ where f_i is the number of distinct values that ever pass through switch s_i .
- (3) Due to network symmetry, half the distinct values passing through a switch follow each of the 2 outputs (we assume here 2 by 2 switches).

Fact: Given these assumptions, the probability of that a value is removed at a switch of level i (measured from the destination end, see network description) is the same for all switches of level i .

According to our assumptions above, at most $2cr$ distinct values pass through a switch connected to a destination processor. Hence, the combining probability at the last switch is $p_1 = \min(1, q/2cr)$. The combining probability at the stage i (measured from destination) is p_i equal to $\min(1, q/(cr(2^{i+1})))$.

A particular value passes through the network without combining, with probability $Pass$ equal to $(1-p_1)(1-p_2) \cdots (1-p_{\log n})$. Therefore, $inmax$ is reduced from cr to $cr + (cr - cr)Pass$.

Non-ideal combining Lemma: Let a be the ratio between the maximum number of distinct $R[C \cup A]$ values arriving at one destination processor, cr , and the memory size, q . If the uniformity assumptions hold then $inmax$ is reduced from cr to $cr + (cr - cr)(1 - \frac{1}{a})$. \square

Example: If $\frac{cr}{q} = 3$, then $\text{Pass} = \frac{2}{3}$ so $\text{inmax} = \frac{cr}{3} + \frac{2crv}{3}$.

So, non-ideal combining is as useful as ideal combining only if the size of the memory is approximately the number of distinct $R[C \cup A]$ values arriving at a single destination processor.

9.3. Analysis of Tagging

Tagging reduces both *outmax* and *inmax* in its first communication step, but requires an extra communication step whose cost is no more than the cost of the first one. (We consider here the semi-join case when *AinattofS* is false. So there is only one extra sending step.)

Tagging Lemma: Under the uniformity assumptions above, tagging reduces *outmax* to $\min(mrv/n, m)$ and *inmax* to $c * \min(n, rv)$. \square

Example: Suppose $r = \frac{|R[C \cup A]|}{|R[C]|} = 10000$, $v = 10$, and $n = 1000$. Combining gives $\text{outmax} = 100m$ and $\text{inmax} = 10000c$. Tagging gives $\text{outmax} = m$ and $\text{inmax} = 1000c$. However, note that tagging requires sending values back, whose effect we can approximate by doubling *inmax* and *outmax*. Note also that we get a degradation by a factor of 10 from optimal speed-up in this case.

9.4. Analysis of Smearing

Intuitively, smearing helps when m is small, causing many values to go to one processor whereas few go to its neighbors. Since smearing requires that S tuples be copied, $|S[C \cup A]|$ should also be small. In this analysis, we assume that $|R|$ is large enough so every $R[C]$ value is in every processor, by the pigeoning lemma.

Using tagging alone in that case, $\text{inmax} = nc$. Suppose that our smearing parameter $k = m/2$. Then for R , $\text{inmax} = \frac{mn}{2k+1}$. (This is actually conservative, because it suggests that all $R[C]$ values hash to an interval of processors of $2k+1$ processors. In the best case, destination processors are more than k processors apart and *inmax* decreases to $\frac{cn}{2k+1}$.)

The communication cost for S however increases. There are two cases: either no tuples of S would have been sent if R were not smeared, in which case C is a key of S and *inmax* and *outmax* increase to $2k|S[C \cup A]|/n$; otherwise, tuples from S were sent and *inmax* and *outmax* increase by a factor of $2k$.

Example: Suppose $|S[C \cup A]| = an$ for some constant a and $|R[C]| = m$. Suppose further that C is the partition key of S . If $k = \frac{m}{2}$ and m is small, then $\text{inmax} = \frac{nm}{m+1} + ma$, whereas by tagging alone $\text{inmax} = n(\log m + 1)$. Thus, smearing only helps if $n \log m > 2am$, which is our decision condition in section 8. This takes into account the fact that *outmax* increases by ma using smearing.

10. Conclusion

We propose and analyze a method for performing joins using symmetric processors interconnected by a high bandwidth network. Our method gives an optimal performance speedup within a constant factor for many cases of the join.

Our analysis suggests that hardware architectural features such as combining can be useful for join processing. Our analysis also suggests that the tagging technique often improves performance when the join operation reduces to a semi-join. On the negative side, our analysis shows that smearing -- a technique for which we had great hope -- is only rarely useful.

The main open problems are to study the system experimentally to gain a better understanding of the constant factors in communication time; to study the implementation and performance issues concerning the maintenance of data structures and of information about the distribution of values in non-prime attributes; and to extend this work to a general query processing strategy.

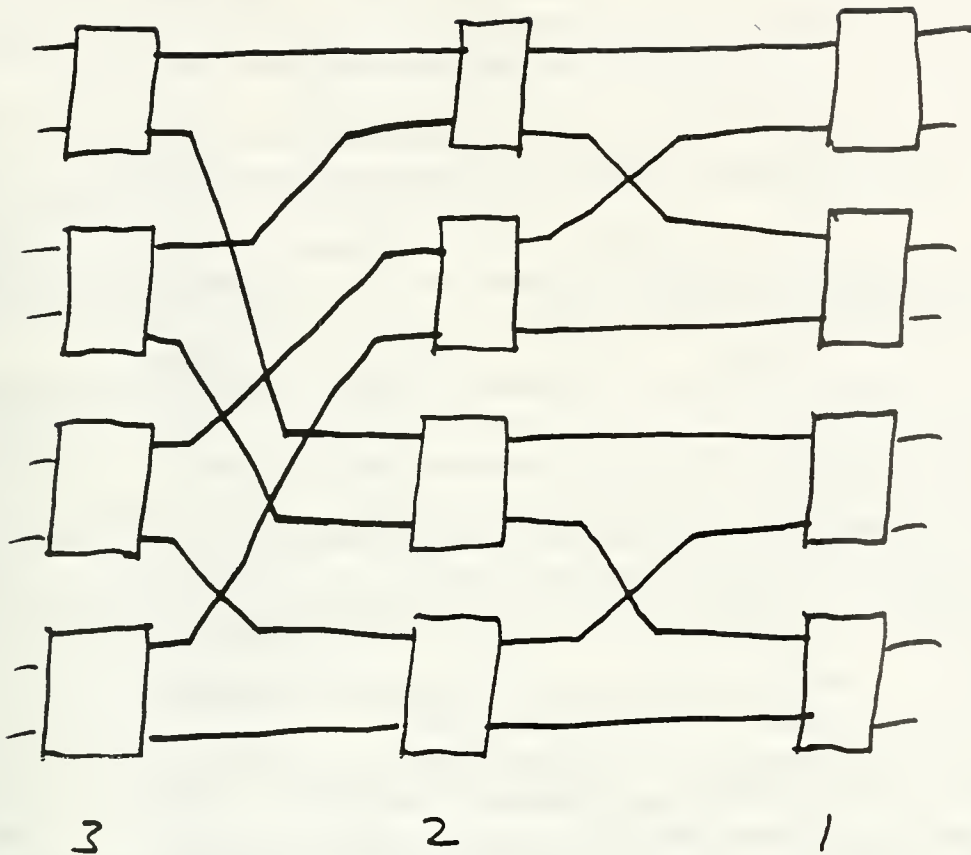


FIGURE 1 BANYON NETWORK FOR EIGHT

PROCESSORS. (PROCESSORS ARE DUPLICATED ON RIGHT.)

IF WE CONSIDER TRANSMISSION PATHS TO GO

FROM LEFT TO RIGHT THEN NUMBERING

CORRESPONDS TO "LEVEL" IN SECTION 6.

REFERENCES

- [B79] E. Babb, "Implementing a Relational Database by Means of Specialized Hardware," ACM TODS 4,1 (March, 1979), 1-29.
- [BC81] P. A. Bernstein and D. W. Chiu, "Using Semijoins to Solve Relational Queries," J. ACM 28:1, pp. 25-40, 1981.
- [BD80] - H. Boral and D. J. DeWitt, "Design considerations for data-flow database machines," in proceedings of the ACM-SIGMOD conference on management of data, 1980, pp. 94-104.
- [BDFW83] - H. Boral, D. J. DeWitt, D. Friedland, and W. K. Wilkinson, "Parallel Algorithms for the execution of relational database operations," ACM Transactions on Database Systems vol. 8, no. 3, September 1983, pp. 324-353.
- [BGWRR81] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie Jr. "Query Processing in a System for Distributed Databases" (SDD-1). ACM Trans. Database Syst. 6, 4 pp. 602-625, 1981.
- [BO79] B. Berra and E. Oliver 1979 "The role of associative array processors in database machine architectures" IEEE Computer, 12, 3, 53-61.
- [CW77] Carter J.L. and Wegman M.N. " Universal classes of hash functions" , Proc. 9th Symposium on Theory of Computing, 1977 , pp 106-112
- [CP] S. Ceri and G. Pelagatti, "Allocation of Operations in Distributed Database Access" IEEE Trans. Comput. C- 31, 2, pp. 119-128.
- [CH82] W. W. Chu and P. Hurley, "Optimal Query Processing for Distributed Database Systems" IEEE Trans. Computing, C-31, 9, pp. 835-850, 1982.
- [D79] D. J. DeWitt, "DIRECT -- a multiprocessor organization for supporting relational database management systems," IEEE Transactions on Computers, C-28, 6, 1979.
- [ES80] R. S. Epstein and M. Stonebraker, "Analysis of query processing strategies for distributed database systems," sixth international conference on very large databases, Montreal, October, 1980.
- [F] H. J. Forker, "Algebraical and operational methods for the optimization of query processing in distributed relational database management systems. In Proceedings of the 2nd International Symposium on Distributed Databases (Berlin, FRG). Elsevier North-Holland, New York, pp. 39-59.
- [G81] Gonnet G.H. , "Expected length of the longest probe sequence in hash code searching" , JACM 28 , 1981 , 289-304.
- [GL73] L. R. Goke and G. J. Lipovsky, "Banyon networks for partitioning multiprocessor systems," in proceedings 1st annual symposium on computer architecture, 1973, pp. 21-28.
- [GS81] J. R. Goodman and C. H. Sequin, "HYPERTREE: a multiprocessor interconnection topology," IEEE Transactions on Computing, 30, 12, 1981.
- [GS82] B. Gavish and A. Segev, "Query Optimization in Distributed Computer Systems" In Management of Distributed Data Processing, J. Akoka, Ed. Elsevier North-Holland, New York, pp. 233-252, 1982.
- [GoodShmu82] - N. Goodman and O. Shmueli, "Tree queries: A simple class of relational queries" ACM Transactions of Database Systems vol. 7, no. 4, December 1982, pp. 653-677.
- [HY79] A. R. Hevner and S. B. Yao, "Query Processing in Distributed Database Systems" IEEE Trans. Softw. Eng. SE-5, 3, pp. 177-187, 1979.

[HY78] A. R. Hevner and S. B. Yao, "Query Processing on a Distributed Database" Proceedings Third Workshop on Distributed Data Management and Computer Networks, August 1978, pp. 91-107.

[H79] D. K. Hsiao, 1979 "Database Machines are Coming, Database Machines are Coming" IEEE Computer 12, 3, pp. 7-9.

[JK84] M. Jarke and J. Koch, "Query Optimization in Database Systems" ACM Computing Surveys, vol. 16, no. 2, June 1984, pp. 111-152.

[KS83] C. P. Kruskal and M. Snir, "The Performance of multistage interconnection networks for multiprocessors," in IEEE transactions on computers, vol. c-32, no. 12, December 1983.

[KTM84] M. Kitsuregawa, H. Tanaka, and T. Moto-oka, "Architecture and Performance of Relational Algebra Machine Grace" IEEE Parallel Processing Conference 1984.

[KTM83] M. Kitsuregawa, H. Tanaka, and T. Moto-oka, "Grace: Relational Algebra Machine Based on Hash and Sort -- Its Design Concepts" Journal of Information Processing, vol 6, no. 3, 1983.

[MH81] M. J. Menon and D. K. Hsiao, "Design and Analysis of a Relational Join Operation for VLSI," Proceedings International Conference on Very Large Database, 1981.

[O82] E. A. Ozkarahan 1982, RAP "Database Machine/Computer Based Distributed Databases." In Proceedings of the 2nd International Symposium on Distributed Databases. (Berlin, FRG). Elsevier North-Holland, New York, pp. 61-80.

[S79] S. Y. W. Su 1979 "Cellular-logic Devices: Concepts and Applications" IEEE Computer 12, 3, 11-25.

[Sch82] M. Schkolnick, "Physical database design techniques." In *Data Base Design Techniques II* S. B. Yao and T. L. Kunii, Eds., Springer-Verlag, pp. 229-252, 1982.

[Sch79] J. W. Schmidt, "Parallel processing of relations: a single-assignment approach." In proceedings of the IEEE 5th international conference on very large data bases, pp. 398-408, 1979.

[SL75] S. Y. W. Su and G. Lipkovsky 1975, "CASSM: A Cellular System for Very Large Databases" In Proceedings of the 1st International Conference on Very Large Data Bases" Framingham, Mass., Sept. 22-24. ACM, New York, pp. 456-472.

[SZ84] R. K. Shultz and R. J. Zingg, "Response Time Analysis of Multiprocessor Computers for Database Support" ACM Transactions of Database Systems, vol. 9, no. 1, March, 1984, pp. 100-132.

[U82] J. D. Ullman, *Principles of Database Systems* second edition. Computer Science Press, 1982.

[VG84] - P. Valduriez and G. Gardarin, "Join and Semijoin Algorithms for a Multiprocessor Database Machine" ACM Transactions of Database Systems, vol. 9, no. 1, March, 1984, pp. 133-161.

[V83] - U. Vishkin, "A parallel-design distributed-implementation (PDDI) general-purpose computer," Technical Report no. 96, New York University department of computer science, June, 1983.

[WY76] E. Wong and K. Youssefi, "Decomposition -- a strategy for query processing" ACM TODS 1, 3 Sept. 1976, pp. 223-241.

This book may be kept

FOURTEEN DAYS

A fine will be charged for each day the book is kept overtime.

JAN 1 1988		
GAYLORD 142		PRINTED IN U S A

NYU COMPSCI TR-158
Shasha, Dennis c.1\

Join processing in a
symmetric ...

NYU COMPSCI TR-158
Shasha, Dennis c.1

Join processing in a
symmetric ...

DATE DUE	BORROWER'S NAME
JAN 11 1988	ET. PEIZONG

LIBRARY
N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012

